Chapter NINE
Lambda Expressions

Exam Objectives

Create and use Lambda expressions.

What is a lambda expression?

In the previous chapter, we saw how passing a piece of code instead of a whole object is very useful.

Java 8 has introduced a notation that enables you to write:

```
List compactCars = findCars(cars,
     (Car c) ->
        car.getType().equals(CarTypes.COMPACT);
```

Instead of:

```
List<Car> compactCars = findCars(cars,
     new Searchable() {
        public boolean test(Car car) {
           return car.getType().equals(
                     CarTypes.COMPACT);
        }
});
```

These are lambda expressions.

The term lambda expression comes from lambda calculus, written as λ-calculus, where λ is the Greek letter lambda. This form of calculus deals with defining and applying functions.

With lambdas, you won't be able to do things that you couldn't do before them, but they allow you to program in a more simple way with a style called functional programming, a different paradigm than object-oriented programming.

A Lambda Expression

```
(Object arg1, int arg2) -> arg1.equals(arg2);
```

A lambda expression has three parts:

A list of parameters

A lambda expression can have zero (represented by empty parentheses), one or more parameters:

```
() -> System.out.println("Hi");
(String s) -> System.out.println(s);
(String s1, String s2) -> System.out.println(s1 + s2);
```

The type of the parameters can be declared explicitly, or it can be inferred from the context:

```
(s) -> System.out.println(s);
```

If there is a single parameter, the type is inferred and it is not mandatory to use parentheses:

```
s -> System.out.println(s);
```

If the lambda expression uses a parameter name which is the same as a variable name of the enclosing context, a compile error is generated:

```
// This doesn't compile
String s = ""; s -> System.out.println(s);
```

An arrow

Formed by the characters `-` and `>` to separate the parameters and the body.

A body

The body of the lambda expressions can contain one or more statements.

If the body has one statement, curly brackets are not required and the value of the expression (if any) is returned:

```
() -> 4; (int a) -> a*6;
```

If the body has more than one statement, curly brackets are required, and if the expression returns a value, it must be returned with a return statement:

```
() -> {
     System.out.println("Hi");
     return 4;
}
(int a) -> {
     System.out.println(a);
     return a*6;
}
```

If the lambda expression doesn't return a result, a `return` statement is optional. For example, the following expressions are equivalent:

```
() -> System.out.println("Hi");
() -> {
     System.out.println("Hi");
     return;
}
```

How are functional interfaces related to all this?

The signature of the abstract method of a functional interface provides the signature of a lambda expression (this signature is called a *functional descriptor*).

This means that to use a lambda expression, you first need a functional interface. For example, using the interface of the previous chapter:

```
interface Searchable {
     boolean test(Car car);
}
```

We can create a lambda expression that takes a `Car` object as argument and returns a `boolean`:

```
Searchable s = (Car c) -> c.getCostUSD() > 20000;
```

In this case, the compiler inferred that the lambda expression can be assigned to a `Searchable` interface, just by its signature.

In fact, lambda expressions don't contain information about which functional interface they are implementing. The type of the expression is deduced from the context in which the lambda is used. This type is called *target type*.

In the previous example, as the lambda expression is being assigned to a `Searchable` interface, the lambda must take the signature of its abstract method. Otherwise, a compiler error is generated.

If we were using the lambda as an argument to a method, the compiler would use the definition of the method to infer the type of the expression:

```
class Test {
     public void find() {
         find(c -> c.getCostUSD() > 20000);
     }
     private void find(Searchable s) {
         // Here goes the implementation
     }
}
```

Because of this, the same lambda expression can be associated with different functional interfaces if they have a compatible abstract method signature. For example:

```
interface Searchable {
     boolean test(Car car);
}
interface Saleable {
     boolean approve(Car car);
}
//...
Searchable s1 = c -> c.getCostUSD() > 20000;
Saleable s2 = c -> c.getCostUSD() > 20000;
```

For reference, the contexts where the target type (the functional interface) of a lambda expression can be inferred are:

- A variable declaration
- An assignment
- A return statement
- An array initializer
- Method or constructor arguments
- A ternary conditional expression
- A cast expression

However, if you understand the concept, you don't need to memorize this list.

But what is it all about functional programming, functional descriptors, functional whatever?

A lambda expression is a function. That's a little different than the methods we know because it's actually related to a *mathematical function*.

A mathematical function takes some inputs and produces some outputs, but **HAS NO SIDE EFFECTS**, meaning that as long as we call it with the same arguments, it always returns the same result.

Of course, in Java, you can't always program in a purely functional style (i.e. without any side effect), only in a *functional-style*.

So for practical terms, it may be safe to think of lambda expressions as anonymous methods (or functions), as they don't have a name, just like anonymous classes.

How are lambda expressions related to anonymous classes?

Lambda expressions are an **ALTERNATIVE** to anonymous classes, but they are not the same.

They have some similarities:

- Local variables (variables or parameters defined in a method) can only be used if they are declared final or are effectively final.
- You can access instance or static variables of the enclosing class.
- They must not throw more exceptions than specified in the throws clause of the functional interface method. Only the same type or a supertype.

And some significant differences:

- For an anonymous class, this keyword resolves to the anonymous class itself. For a lambda expression, it resolves to the enclosing class where the lambda is written.
- Default methods of a functional interface cannot be accessed from within lambda expressions. Anonymous classes can.
- Anonymous classes are compiled into, well, inner classes. But lambda expressions are converted into private static (in some cases) methods of their enclosing class and, using the `invokedynamic` instruction (added in Java 7), they are bound dynamically. Since there's no need to load another class, lambda expressions are more efficient. This is a very simple explanation, but that's the idea.

By the way, if you really have to know or want to understand why local variables have to be final whereas instance or static variables don't when using them inside an anonymous class or lambda expression, it's because of the way these variables are implemented in Java.

Instance variables are stored on the heap (a region of your computer's memory accessible anywhere in your program), while local variables live on the stack (a special region of your computer's memory that stores temporary variables created by each function).

Variables on the heap are shared by across threads, but variables on the stack are implicitly confined to the thread they're in.

So when you create an instance of an anonymous inner class or a lambda expression, the values of local variables are **COPIED**. In other words, you're not working with the variable, but with a copied value.

First of all, this prevents thread-related problems. Second, since you're working with a copy, if the variable could be modified by the rest of the method, you would be working with an out-of-date variable.

A `final` (or effectively final) variable removes these possibilities. Since the value can't be changed, you don't need to worry about such changes being visible or causing thread problems.

Key Points

- Lambda expressions have three parts: a list of parameters, and arrow, and a body:
  ```
  (Object o) -> System.out.println(o);
  ```
- You can think of lambda expressions as anonymous methods (or functions) as they don't have a name.
- A lambda expression can have zero (represented by empty parentheses), one or more parameters.
- The type of the parameters can be declared explicitly, or it can be inferred from the context.
- If there is a single parameter, the type is inferred and is not mandatory to use parentheses.
- If the lambda expression uses as a parameter name which is the same as a variable name of the enclosing context, a compile error is generated.
- If the body has one statement, curly brackets are not required, and the value of the expression (if any) is returned.
- If the body has more than one statement, curly brackets are required, and if the expression returns a value, it must return with a `return` statement.
- If the lambda expression doesn't return a result, a `return` statement is optional.
- The signature of the `abstract method` of a functional interface provides the signature of a lambda expression (this signature is called a *functional descriptor*).
- This means that to use a lambda expression, you first need a functional interface.
- However, lambda expressions don't contain the information about which functional interface are implementing.
- The type of the expression is deduced from the context in which the lambda is used. This type is called *target type*.
- The contexts where the target type of a lambda expression can be inferred include an assignment, method or constructor arguments, and a cast expression.
- Like anonymous classes, lambda expressions can access instance and static variables, but only final or effectively final local variables.

- Also, they cannot throw exceptions that are not defined in the `throws` clause of the function interface method.
- For a lambda expression, this resolves to the enclosing class where the lambda is written.
- Default methods of a functional interface cannot be accessed from within lambda expressions.

Self Test

1. Which of the following are valid lambda expressions?
A. `String a, String b -> System.out.print(a+ b);`
B. `() -> return;`
C. `(int i) -> i;`
D. `(int i) -> i++; return i;`

2. Given:

```
interface A {
     int aMethod(String s);
}
```

Which of the following are valid statements?
A. `A a = a -> a.length();`
B. `A x = y -> {return y;};`
C. `A s = "2" -> Integer.parseInt(s);`
D. `A b = (String s) -> 1;`

3. A lambda expression can be used...
A. As a method argument
B. As a conditional expression in an `if` statement
C. In a `return` statement
D. In a `throw` statement

4. Given:

```
() -> 7 * 12.0;
```

Which of the following interfaces can provide the functional descriptor for the above lambda expression?
A.

```
interface A {
     default double m() {
         return 4.5;
     }
}
```

B.

```
interface B {
     Number m();
}
```

C.

```
interface C {
     int m();
```

```
}
```

D.

```
interface D {
     double m(Integer... i);
}
```

5. Given:

```
interface AnInterface {
     default int aMethod() { return 0; }
     int anotherMethod();
}
public class Question_9_5 implements AnInterface {
     public static void main(String[] args) {
         AnInterface a = () -> aMethod();
         System.out.println(a.anotherMethod());
     }
     @Override
     public int anotherMethod() {
         return 1;
     }
}
```

What is the result?
A. `0`
B. `1`
C. Compilation fails
D. An exception occurs at runtime

6. Which of the following statements are true?
A. Curly brackets are required whenever the `return` keyword is used in a lambda expression
B. A `return` keyword is always required in a lambda expression
C. A `return` keyword is always optional in a lambda expression
D. Lambda expressions don't return values

7. How is the `this` keyword handled inside a lambda expression?
A. You can't use this inside a lambda expression
B. `this` refers to the functional interface of the lambda expression
C. `this` refers to the lambda expression itself
D. `this` refers to the enclosing class of the lambda expression

8. Given:

```
interface X {
     int test(int i);
}
public class Question_9_8 {
     int i = 0;
     public static void main(String[] args) {
         X x = i -> i * 2;
         System.out.println(x.test(3));
     }
}
```

What is the result?
A. `0`
B. `3`

C. 6
D. Compilation fails

Open answers page

Do you like what you read? Would you consider?

Buying the print/kindle version from Amazon

Buying the PDF/EPUB/MOBI versions from Leanpub

Buying the e-book version from iTunes

Buying the e-book version from Kobo

Buying the e-book version from Scribd

Do you have a problem or something to say?

Report an issue with the book

Contact me

---

08. Lambda Expressions 10. Java Built-In Lambda Interfaces